

UTN - TUP

Trabajo Practico Integrador

Alumnos:

Ramallo, Geronimo Gaston

Lahoz, Cristian Daniel

Materia: Programación I

Profesor: Bruselario, Sebastián

Tutor: Cimino, Virginia

JUNIO 2025

Introducción

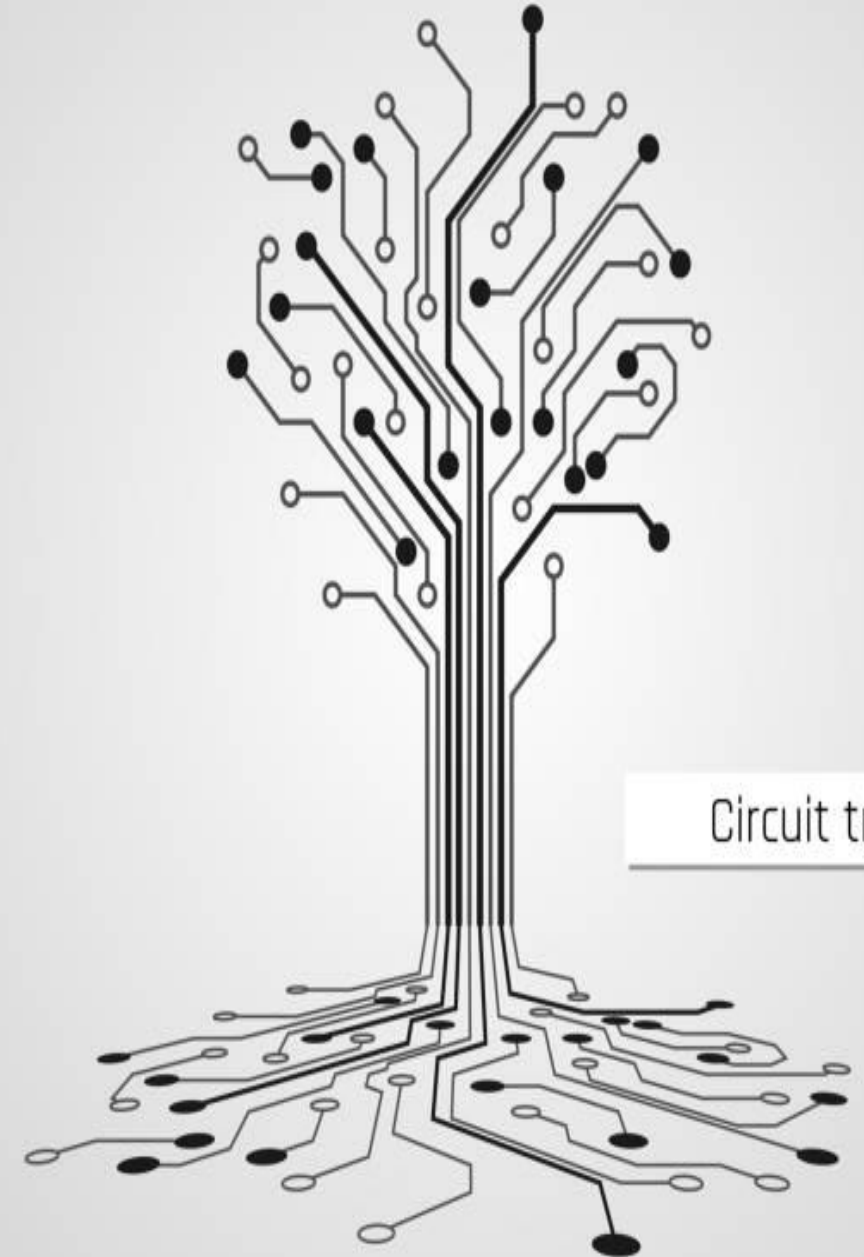
Las listas anidadas proporcionan una forma intuitiva de representar la estructura jerárquica de los árboles sin requerir conocimientos avanzados de Programación Orientada a Objetos (POO).

Los árboles son esenciales en múltiples aplicaciones como:

- Sistemas de archivos
- Bases de datos
- Algoritmos de búsqueda y ordenamiento
- Inteligencia Artificial (árboles de decisión)

Objetivos

- Implementar las operaciones básicas de árboles binarios usando listas anidadas
- Desarrollar funciones para visualización y recorrido
- Analizar la eficiencia de esta representación
- Demostrar aplicaciones prácticas de esta estructura

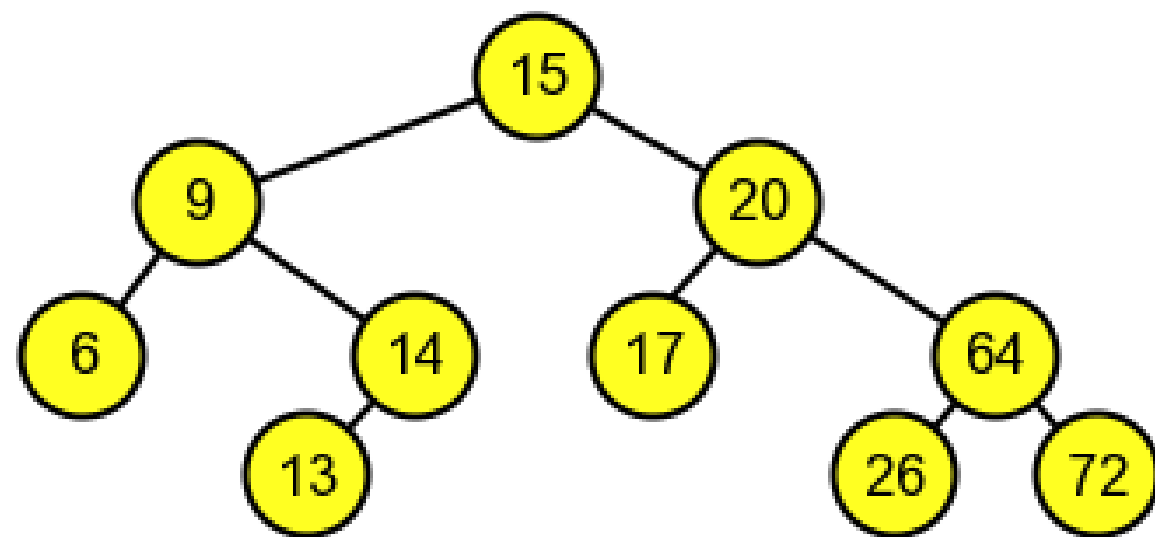


Marco Teórico

Árbol Binario

Un árbol es una estructura de datos no lineal en la que cada nodo puede apuntar a uno o varios nodos. Sin embargo, no tienen una estructura lógica de tipo lineal o secuencial, sino ramificada. Tienen aspecto de árbol, de ahí su nombre.

Un caso particular es el árbol binario, en el que ningún nodo puede tener más de dos subárboles. Cada nodo puede tener cero, uno o dos hijos. Se conoce el nodo de la izquierda como hijo izquierdo y el nodo de la derecha como hijo derecho.



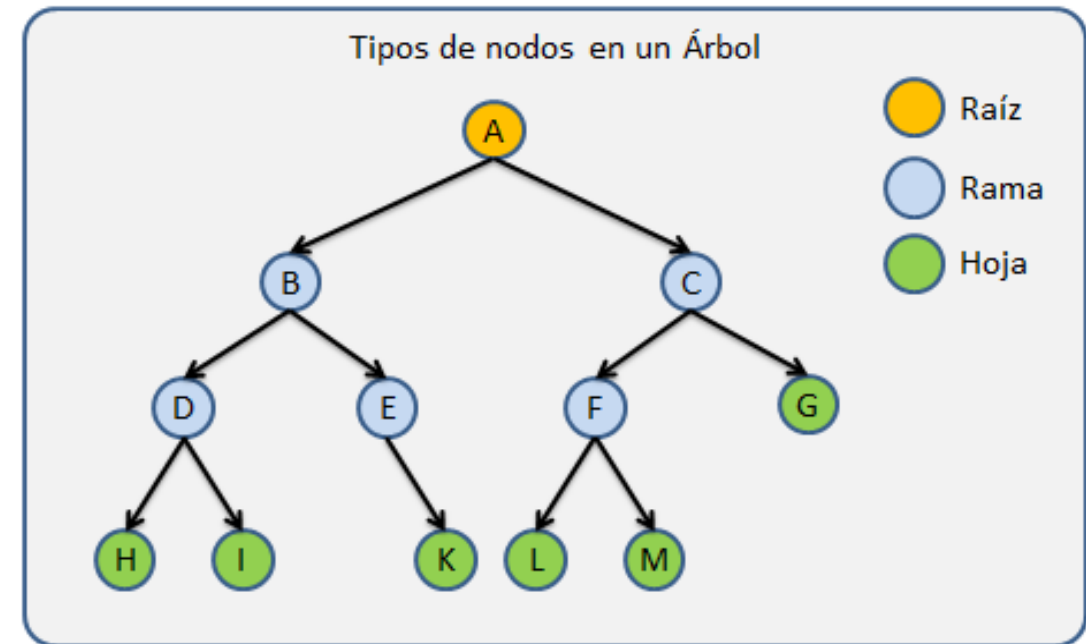
Clasificación de Nodos

Según su ubicación en el árbol

- *Nodo raíz*: Punto de entrada a la estructura. No tiene un nodo padre y es único.
- *Nodo rama o interno*: Nodos intermedios entre la raíz y las hojas. Tiene un padre y, al menos, un hijo.
- *Nodos hoja*: Nodos sin hijos. Están en los extremos del árbol.

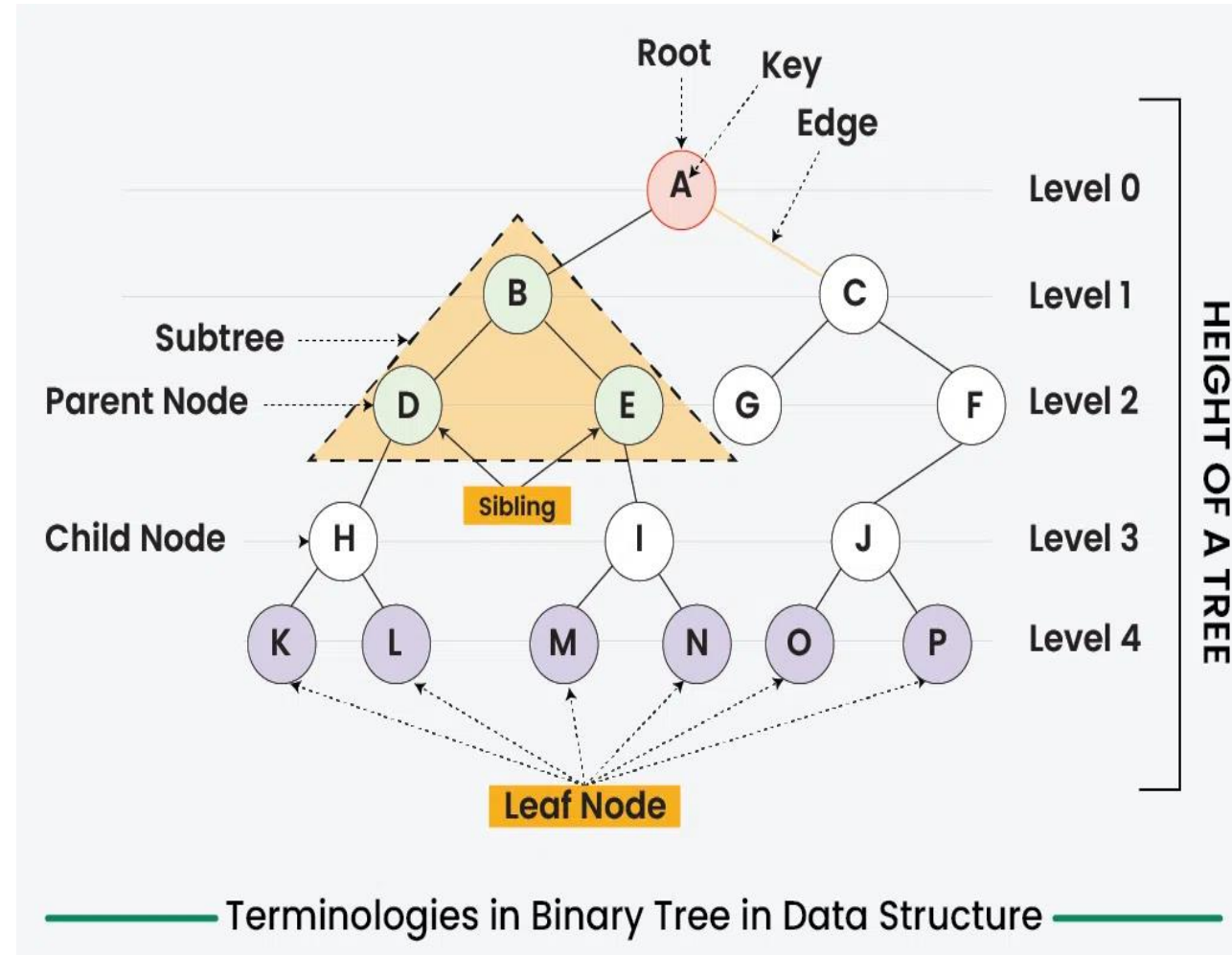
Según su relación con otros nodos

- *Nodo padre* : Tiene, al menos, un hijo conectado a él.
- *Nodo hijo*: Está conectado con un Nodo padre, se encuentra debajo de él en la jerarquía.
- *Nodo hermano*: Comparte el mismo Nodo padre, se encuentran en el mismo nivel de la jerarquía.



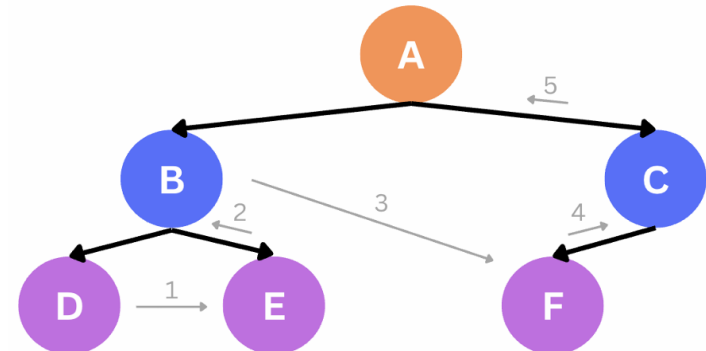
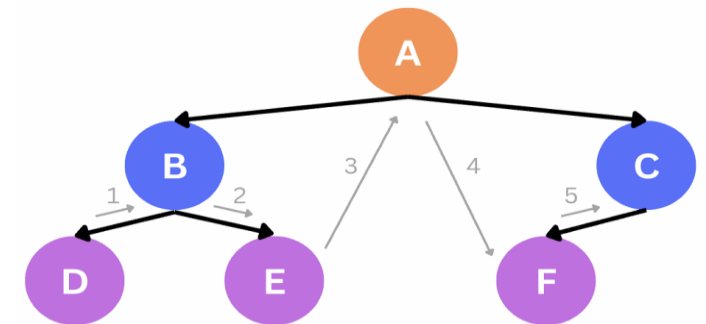
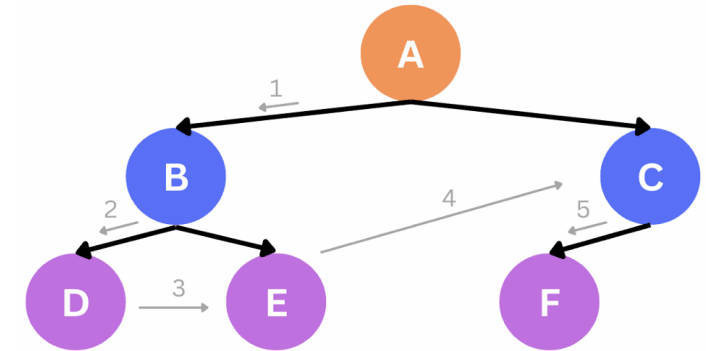
Propiedades Clave

- *Altura*: Longitud del camino más largo desde la raíz hasta una hoja
- *Nivel*: Longitud del camino que lo conecta al Nodo raíz
- *Grado*: Número máximo de hijos por nodo (2 en árboles binarios)
- *Orden*: Cantidad máxima de hijos que puede tener un Nodo padre. Es una condición restrictiva para la construcción del árbol
- *Peso*: número total de Nodos que tiene un árbol



Recorridos

- Preorden: la raíz se recorre antes que los recorridos de los subárboles izquierdo y derecho.
- Inorden: la raíz se recorre entre los recorridos de los árboles izquierdo y derecho.
- Postorden: la raíz se recorre después de los recorridos por el subárbol izquierdo y el derecho.



Representación con Listas Anidadas

En Python, podemos representar árboles binarios mediante listas anidadas con el formato:

```
[valor, subarbol_izquierdo, subarbol_derecho]
```

Donde:

- valor es el dato almacenado en el nodo
- subarbol_izquierdo es la lista que representa el hijo izquierdo ([] si no existe)
- subarbol_derecho es la lista que representa el hijo derecho ([] si no existe)

```
1 arbol_ejemplo: list = [  
2     "A",  
3     [  
4         "B",  
5         [  
6             "D",  
7             ["H", [], []],  
8             ["I", [], []]  
9         ],  
10        ["E", [], []]  
11    ],  
12    [  
13        "C",  
14        ["F", [], []],  
15        ["G", [], []]  
16    ]  
17 ]
```

Operaciones Básicas


- *Creación:*
 - Construcción manual de la lista anidada.
 - Función auxiliar para crear nodos.
- *Inserción:*
 - Agregar nuevos nodos manteniendo la estructura.
- *Recorrido:*
 - Implementación recursiva de los tres tipos de recorrido.
- *Visualización:*
 - Representación gráfica del árbol en consola.



Caso Práctico

Implementación de Operaciones con Árboles
Desarrollamos un módulo Python que implementa:

Estructura básica



```
1 def crear_nodo_raiz(valor: str) → list:
2     """
3     Crea y retorna un nodo raíz con el valor proporcionado.
4
5     El nodo se representa como una lista de tres elementos:
6     [valor, hijo_izquierdo, hijo_derecho], donde ambos hijos
7     son listas vacías al inicio.
8
9     Args:
10         valor (str): Valor que contendrá el nodo raíz.
11
12     Returns:
13         list: Nodo raíz representado como lista anidada.
14     """
15
16     return [valor, [], []]
```

Inserción

```
1 def insertar_hijo_izquierdo(arbol: list, valor: str) → list:
2     """
3     Inserta un nuevo nodo como hijo izquierdo del nodo actual.
4
5     Si ya existe un hijo izquierdo, este pasará a ser hijo izquierdo
6     del nuevo nodo insertado.
7
8     Args:
9         arbol (list): Árbol o subárbol donde se insertará el hijo izquierdo.
10        valor (str): Valor del nuevo nodo.
11
12    Returns:
13        list: Árbol actualizado con el nuevo nodo insertado.
14    """
15    if not arbol:
16        print("Árbol vacío")
17        return
18    subarbol_izq: list = arbol.pop(1)
19    arbol.insert(1, [valor, subarbol_izq, []])
20    return arbol
21
22
23 def insertar_hijo_derecho(arbol: list, valor: str) → list:
24     """
25     Inserta un nuevo nodo como hijo derecho del nodo actual.
26
27     Si ya existe un hijo derecho, este pasará a ser hijo derecho
28     del nuevo nodo insertado.
29
30     Args:
31         arbol (list): Árbol o subárbol donde se insertará el hijo derecho.
32         valor (str): Valor del nuevo nodo.
33
34    Returns:
35        list: Árbol actualizado con el nuevo nodo insertado.
36    """
37    if not arbol:
38        print("Árbol vacío")
39        return
40    subarbol_der: list = arbol.pop(2)
41    arbol.insert(2, [valor, [], subarbol_der])
42    return arbol
```

Recorridos

08/06/2025

```
1 def recorrer_preorden(arbol: list) → None:
2     """
3     Realiza un recorrido en preorden del árbol (raíz → izq → der)
4     e imprime los valores de los nodos en ese orden.
5
6     Args:
7         arbol (list): Árbol binario representado con listas anidadas.
8
9     Returns:
10        None: Llama recursivamente a la función para recorrer el árbol.
11    """
12    if arbol:
13        print(arbol[0], end=" ")
14        recorrer_preorden(arbol[1])
15        recorrer_preorden(arbol[2])
16
17
18 def recorrer_inorden(arbol: list) → None:
19     """
20     Realiza un recorrido en inorden del árbol (izq → raíz → der)
21     e imprime los valores de los nodos en ese orden.
22
23     Args:
24         arbol (list): Árbol binario representado con listas anidadas.
25
26     Returns:
27        None: Llama recursivamente a la función para recorrer el árbol.
28    """
29    if arbol:
30        recorrer_inorden(arbol[1])
31        print(arbol[0], end=" ")
32        recorrer_inorden(arbol[2])
33
34
35 def recorrer_postorden(arbol: list) → None:
36     """
37     Realiza un recorrido en postorden del árbol (izq → der → raíz)
38     e imprime los valores de los nodos en ese orden.
39
40     Args:
41         arbol (list): Árbol binario representado con listas anidadas.
42
43     Returns:
44        None: Llama recursivamente a la función para recorrer el árbol.
45    """
46    if arbol:
47        recorrer_postorden(arbol[1])
48        recorrer_postorden(arbol[2])
49        print(arbol[0], end=" ")
```

Altura y Peso

08/06/2025

```
1 def altura_arbol(arbol: list) → int:
2     """
3     Calcula la altura máxima de un árbol binario representado con listas anidadas.
4
5     La altura de un árbol se define como el número de aristas en el camino más largo
6     desde el nodo raíz hasta una hoja. Un árbol vacío tiene altura 0.
7
8     Args:
9         arbol (list): Árbol binario representado como lista anidada.
10
11     Returns:
12         int: Altura del árbol. Retorna 0 para árbol vacío o solo con raíz.
13     """
14     # Caso base 1: árbol vacío
15     if not arbol:
16         return 0 # Altura 0 para árbol vacío por convención en esta implementación
17
18     # Caso recursivo: 1 (nodo actual) + máximo entre alturas de subárboles
19     return 1 + max(
20         altura_arbol(arbol[1]), # Altura subárbol izquierdo
21         altura_arbol(arbol[2]), # Altura subárbol derecho
22     )
23
24
25 def peso_arbol(arbol: list) → int:
26     """
27     Calcula el número total de nodos en un árbol binario representado con listas anidadas.
28
29     El peso de un árbol es la cantidad total de nodos que contiene.
30     Un árbol vacío tiene peso 0.
31
32     Args:
33         arbol (list): Lista que representa el árbol en formato [valor, hijo_izquierdo, hijo_derecho]
34         o lista vacía [] para árbol vacío.
35
36     Returns:
37         int: Cantidad total de nodos en el árbol.
38     """
39     # Caso base: árbol vacío
40     if not arbol:
41         return 0
42
43     # Caso recursivo: 1 (nodo actual) + peso subárbol izquierdo + peso subárbol derecho
44     return 1 + peso_arbol(arbol[1]) + peso_arbol(arbol[2])
```

Visualización

```
1 def visualizar_arbol(
2     arbol: list,
3     prefijo: str = "",
4     es_ultimo: bool = True,
5     es_raiz: bool = True,
6 ) → None:
7     """
8     Visualiza un árbol binario de manera estructurada.
9
10    Args:
11        arbol (list): Lista que representa el árbol [valor, izquierdo, derecho]
12        prefijo (str): Prefijo para la indentación
13        es_ultimo (bool): Si es el último hijo de su padre
14        es_raiz (bool): Si es el nodo raíz
15
16    Returns:
17        None: Llama recursivamente a la función para visualizar el árbol.
18    """
19
20    # Constantes con caracteres Unicode para la visualización
21    RAMA = "├─"
22    ULTIMO = "└─"
23    VERTICAL = "│"
24    ESPACIO = "  "
25
26    # Si el árbol existe
27    if arbol:
28        # Mostrar el nodo actual
29        if es_raiz:
30            # Si es el nodo raíz, imprimirlo sin prefijo ni decoración
31            print(arbol[0])
32
33            # Reiniciar el prefijo para los hijos de la raíz
34            nuevo_prefijo: str = ""
35        else:
36            # Para nodos no raíz:
37            # 1. Imprimir el prefijo acumulado (indentación)
38            # 2. Usar ULTIMO (└─) si es el último hijo de su padre, RAMA (├─) si no
39            # 3. Añadir el valor del nodo actual
40            print(prefijo + (ULTIMO if es_ultimo else RAMA) + str(arbol[0]))
41
42            # Calcular nuevo prefijo para los hijos:
43            # - Si es último hijo: añadir espacios en blanco ("  ")
44            # - Si no: añadir barra vertical ("│ ") para conectar visualmente
45            nuevo_prefijo: str = prefijo + (ESPACIO if es_ultimo else VERTICAL)
46
47            # Obtener el hijo derecho (se procesa primero para efecto espejo)
48            hijo_primero: list = arbol[2]
49
50            # Obtener el hijo izquierdo
51            hijo_segundo: list = arbol[1]
52
53            # Procesar hijos en el orden determinado
54            hijos: list = []
55
56            # Si existe hijo derecho
57            if hijo_primero:
58                hijos.append(hijo_primero)
59
60            # Si existe hijo izquierdo
61            if hijo_segundo:
62                hijos.append(hijo_segundo)
63
64            # Recorrer todos los hijos para procesarlos recursivamente
65            for i, hijo in enumerate(hijos):
66                # Determinar si es el último hijo para usar el conector adecuado
67                ultimo_hijo: bool = i == len(hijos) - 1
68
69                # Llamada recursiva
70                visualizar_arbol(hijo, nuevo_prefijo, ultimo_hijo, False)
```

Ejemplo de Uso

```
1  # Crear un árbol binario de ejemplo
2  arbol = crear_nodo_raiz("A")
3  insertar_hijo_izquierdo(arbol, "B")
4  insertar_hijo_derecho(arbol, "C")
5  insertar_hijo_izquierdo(arbol[1], "D")
6  insertar_hijo_derecho(arbol[1], "E")
7  insertar_hijo_izquierdo(arbol[2], "F")
8  insertar_hijo_derecho(arbol[2], "G")
9  insertar_hijo_izquierdo(arbol[1][1], "H")
10 insertar_hijo_derecho(arbol[1][1], "I")
11
12 # Mostrar el árbol y sus recorridos
13 print("\n=== VISUALIZACIÓN DEL ÁRBOL ===\n")
14 visualizar_arbol(arbol)
15 print("\n=== RECORRIDOS ===")
16 print("\nPreorden:")
17 recorrer_preorden(arbol)
18 print("\nInorden:")
19 recorrer_inorden(arbol)
20 print("\nPostorden:")
21 recorrer_postorden(arbol)
22 print("\n\n=== PROPIEDADES ===")
23 print(f"\nAltura del árbol → {altura_arbol(arbol)}")
24 print(f"Peso del árbol → {peso_arbol(arbol)}")
```

Metodología Utilizada

Investigación Preliminar

- Revisión de documentación sobre estructuras de datos
- Análisis de implementaciones alternativas

Diseño

- Definición de la estructura de datos
- Planificación de funciones principales
- Diseño de interfaz de usuario (consola)

Implementación

- Desarrollo iterativo de funciones
- Refinamiento basado en resultados

Herramientas

- Python 3.11
- Visual Studio Code
- Control de versiones con Git/GitHub

Trabajo colaborativo

- División de tareas por módulos
- Revisiones de código cruzadas
- Integración continua

Resultados Obtenidos

Implementación exitosa de todas las operaciones básicas

- Creación e inserción de nodos
- Recorridos preorden, inorden y postorden
- Visualización jerárquica

Dificultades superadas

- Manejo de recursión en recorridos
- Formato de visualización

Caso de prueba

- Árbol de 4 niveles con 9 nodos

Repositorio GitHub

- Código completo documentado
- Ejemplos de uso
- Instrucciones de instalación

Conclusiones

Aprendizajes

- Dominio de la representación de árboles con listas
- Manejo avanzado de recursión en Python
- Importancia de la visualización en estructuras complejas

Aplicaciones

- Base para estructuras más complejas (árboles AVL, Heap)
- Uso en algoritmos de búsqueda y ordenamiento
- Modelado de problemas jerárquicos

Mejoras futuras

- Implementación de balanceo automático
- Serialización/deserialización del árbol
- Interfaz gráfica para visualización

Trabajo en equipo

- Coordinación efectiva en desarrollo distribuido
- Revisión cruzada de código
- Integración de contribuciones